

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Lazy Loading with Code Conversion

Inventors:

Debi Mishra

Sushil Baid

Ramesh Chandrasekhar

ATTORNEY'S DOCKET NO. MS1-927US

1 Lazy Loading with Code Conversion

2 RELATED APPLICATIONS

3 This application is copending with application no. _____, entitled
4 "Semantics Mapping Between Different Object Hierarchies", filed concurrently
5 herewith, by Mishra et al.(MS1-929US) and is incorporated herein by reference.
6

7 TECHNICAL FIELD

8 The subject matter relates generally to methods and/or devices for
9 enhancing portability of programming language codes and processed codes, in
10 particular, through methods and/or devices related to loading of code during
11 runtime.
12

13 BACKGROUND

14 An object-oriented programming language (OOPL) typically defines not
15 only the data type of a data structure, but also the types of functions that can be
16 applied to the data structure. In essence, the data structure becomes an object that
17 includes both data and functions. During execution of an OOPL program, access
18 to an object's functionality occurs by calling its methods and accessing its
19 properties, events, and/or fields.
20

21 In an OOPL environment, objects are often divided into classes wherein
22 objects that are an instance of the same class share some common property or
23 properties (e.g., methods and/or instance variables). Relationships between
24 classes form a class hierarchy, also referred to herein as an object hierarchy.
25 Through this hierarchy, objects can inherit characteristics from other classes.

1
2 In object-oriented programming, the terms "Virtual Machine" (VM) and
3 "Runtime Engine" (RE) have recently become associated with software that
4 executes code on a processor or a hardware platform. In the description presented
5 herein, the term "RE" includes VM. A RE often forms part of a larger system or
6 framework that allows a programmer to develop an application for a variety of
7 users in a platform independent manner. For a programmer, the application
8 development process usually involves selecting a framework, coding in an OOPL
9 associated with that framework, and compiling the code using framework
10 capabilities. The resulting platform-independent, compiled code is then made
11 available to users, usually as an executable file and typically in a binary format.
12 Upon receipt of an executable file, a user can execute the application on a RE
13 associated with the selected framework.
14

15 Traditional frameworks, such as the JAVATM language framework (Sun
16 Microsystems, Inc., Palo Alto, California), were developed initially for use with a
17 single OOPL (i.e., monolithic at the programming language level); however, a
18 recently developed framework, .NETTM framework (Microsoft Corporation,
19 Redmond, Washington), allows programmers to code in a variety of OOPLs. This
20 multi-OOPL framework is centered around a single compiled "intermediate"
21 language having a virtual object system (VOS). As a result, the object hierarchy
22 and the nature of the compiled code differ between the JAVATM language
23 framework and the .NETTM framework.
24
25

For the discussion presented herein, the term “bytecode” is generally associated with a first framework and the term “intermediate language code” or “IL code” is associated with a second framework, typically capable of compiling a variety of programming languages. In a typical framework, the framework RE compiles code to platform-specific or “native” machine. This second compilation produces an executable native machine code. Throughout the following description, a distinction is drawn between the first compilation process (which compiles a programming language code to bytecode or an intermediate language code) and the second compilation process (which compiles, for example, a bytecode or an intermediate language code to native machine code/instructions). In general, a “compiled code” (or “compiled codes”) refers to the result of the first compilation process.

To enhance portability of programming languages and compiled codes, there is a need for methods and/or devices that can perform the following acts: (i) compile a programming language code associated with a first framework (e.g., a bytecode framework) to a compiled code associated with a second framework (e.g., an IL code framework); and (ii) convert a compiled code associated with a first framework (e.g., a bytecode framework) to a compiled code associated with a second framework (e.g., an IL code framework). Such methods and/or devices should account for differences in class loading and perform without substantially compromising the original programmer’s intent.

SUMMARY

1 Exemplary devices and/or methods optionally compile a programming
2 language code associated with one framework to a code associated with another
3 framework; and/or convert a code associated with one framework to a code
4 associated with another framework. The aforementioned devices and/or methods
5 optionally include, but are not limited to, features for supporting framework
6 differences in object hierarchy, exceptions, type characteristics, reflection
7 transparency, and/or scoping, and features for supporting differences in class
8 loading.

9
10 An exemplary method receives an initial code associated with a first
11 framework, the initial code including a reference to a referenced class; converts
12 the initial code to a converted code capable of execution on a second framework;
13 executes the converted code on the second framework; detects a need for the
14 referenced class during execution of the converted code on the second framework;
15 and loads the referenced class into memory assessable by the second framework.

16
17 Additional features and advantages of the invention will be made apparent
18 from the following detailed description of illustrative embodiments, which
19 proceeds with reference to the accompanying figures.

20 21 **BRIEF DESCRIPTION OF THE DRAWINGS**

22 A more complete understanding of the various methods and arrangements
23 described herein, and equivalents thereof, may be had by reference to the
24 following detailed description when taken in conjunction with the accompanying
25 drawings wherein:

1 Fig. 1 is a block diagram generally illustrating an exemplary computer
2 system on which the exemplary methods and exemplary systems described herein
3 may be implemented.

4 Fig. 2 is a block diagram illustrating a Web site/server and an exemplary
5 user system capable of receiving, converting and executing code from the Web
6 site/server.

7 Fig. 3 is a block diagram illustrating an exemplary method for receiving,
8 converting and executing code.

9 Fig. 4 is a block diagram illustrating an exemplary method for receiving,
10 converting and executing code.

11 12 **DETAILED DESCRIPTION**

13 Turning to the drawings, wherein like reference numerals refer to like
14 elements, various methods and converters are illustrated as being implemented in a
15 suitable computing environment. Although not required, the methods and
16 converters will be described in the general context of computer-executable
17 instructions, such as program modules, being executed by a personal computer.
18 Generally, program modules include routines, programs, objects, components, data
19 structures, etc. that perform particular tasks or implement particular abstract data
20 types. Moreover, those skilled in the art will appreciate that the methods and
21 converters may be practiced with other computer system configurations, including
22 hand-held devices, multi-processor systems, microprocessor based or
23 programmable consumer electronics, network PCs, minicomputers, mainframe
24 computers, and the like. The methods and converters may also be practiced in
25 distributed computing environments where tasks are performed by remote

1 processing devices that are linked through a communications network. In a
2 distributed computing environment, program modules may be located in both local
3 and remote memory storage devices.

4
5 Fig.1 illustrates an example of a suitable computing environment 120 on
6 which the subsequently described methods and converter arrangements may be
7 implemented.

8
9 Exemplary computing environment 120 is only one example of a suitable
10 computing environment and is not intended to suggest any limitation as to the
11 scope of use or functionality of the improved methods and arrangements described
12 herein. Neither should computing environment 120 be interpreted as having any
13 dependency or requirement relating to any one or combination of components
14 illustrated in computing environment 120.

15
16 The improved methods and arrangements herein are operational with
17 numerous other general purpose or special purpose computing system
18 environments or configurations. Examples of well known computing systems,
19 environments, and/or configurations that may be suitable include, but are not
20 limited to, personal computers, server computers, thin clients, thick clients, hand-
21 held or laptop devices, multiprocessor systems, microprocessor-based systems, set
22 top boxes, programmable consumer electronics, network PCs, minicomputers,
23 mainframe computers, distributed computing environments that include any of the
24 above systems or devices, and the like.

1 As shown in Fig. 1, computing environment 120 includes a general-purpose
2 computing device in the form of a computer 130. The components of computer
3 130 may include one or more processors or processing units 132, a system
4 memory 134, and a bus 136 that couples various system components including
5 system memory 134 to processor 132.

6
7 Bus 136 represents one or more of any of several types of bus structures,
8 including a memory bus or memory controller, a peripheral bus, an accelerated
9 graphics port, and a processor or local bus using any of a variety of bus
10 architectures. By way of example, and not limitation, such architectures include
11 Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA)
12 bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA)
13 local bus, and Peripheral Component Interconnects (PCI) bus also known as
14 Mezzanine bus.

15
16 Computer 130 typically includes a variety of computer readable media.
17 Such media may be any available media that is accessible by computer 130, and it
18 includes both volatile and non-volatile media, removable and non-removable
19 media.

20
21 In Fig. 1, system memory 134 includes computer readable media in the
22 form of volatile memory, such as random access memory (RAM) 140, and/or non-
23 volatile memory, such as read only memory (ROM) 138. A basic input/output
24 system (BIOS) 142, containing the basic routines that help to transfer information
25 between elements within computer 130, such as during start-up, is stored in ROM

1 138. RAM 140 typically contains data and/or program modules that are
2 immediately accessible to and/or presently being operated on by processor 132.

3
4 Computer 130 may further include other removable/non-removable,
5 volatile/non-volatile computer storage media. For example, Fig. 1 illustrates a
6 hard disk drive 144 for reading from and writing to a non-removable, non-volatile
7 magnetic media (not shown and typically called a "hard drive"), a magnetic disk
8 drive 146 for reading from and writing to a removable, non-volatile magnetic disk
9 148 (e.g., a "floppy disk"), and an optical disk drive 150 for reading from or
10 writing to a removable, non-volatile optical disk 152 such as a CD-ROM, CD-R,
11 CD-RW, DVD-ROM, DVD-RAM or other optical media. Hard disk drive 144,
12 magnetic disk drive 146 and optical disk drive 150 are each connected to bus 136
13 by one or more interfaces 154.

14
15 The drives and associated computer-readable media provide nonvolatile
16 storage of computer readable instructions, data structures, program modules, and
17 other data for computer 130. Although the exemplary environment described
18 herein employs a hard disk, a removable magnetic disk 148 and a removable
19 optical disk 152, it should be appreciated by those skilled in the art that other types
20 of computer readable media which can store data that is accessible by a computer,
21 such as magnetic cassettes, flash memory cards, digital video disks, random access
22 memories (RAMs), read only memories (ROM), and the like, may also be used in
23 the exemplary operating environment.

1 A number of program modules may be stored on the hard disk, magnetic
2 disk 148, optical disk 152, ROM 138, or RAM 140, including, e.g., an operating
3 system 158, one or more application programs 160, other program modules 162,
4 and program data 164.

5
6 The improved methods and arrangements described herein may be
7 implemented within operating system 158, one or more application programs 160,
8 other program modules 162, and/or program data 164.

9
10 A user may provide commands and information into computer 130 through
11 input devices such as keyboard 166 and pointing device 168 (such as a "mouse").
12 Other input devices (not shown) may include a microphone, joystick, game pad,
13 satellite dish, serial port, scanner, camera, etc. These and other input devices are
14 connected to the processing unit 132 through a user input interface 170 that is
15 coupled to bus 136, but may be connected by other interface and bus structures,
16 such as a parallel port, game port, or a universal serial bus (USB).
17

18 A monitor 172 or other type of display device is also connected to bus 136
19 via an interface, such as a video adapter 174. In addition to monitor 172, personal
20 computers typically include other peripheral output devices (not shown), such as
21 speakers and printers, which may be connected through output peripheral interface
22 175.

23
24 Logical connections shown in Fig. 1 are a local area network (LAN) 177
25 and a general wide area network (WAN) 179. Such networking environments are

1 commonplace in offices, enterprise-wide computer networks, intranets, and the
2 Internet.

3
4 When used in a LAN networking environment, computer 130 is connected
5 to LAN 177 via network interface or adapter 186. When used in a WAN
6 networking environment, the computer typically includes a modem 178 or other
7 means for establishing communications over WAN 179. Modem 178, which may
8 be internal or external, may be connected to system bus 136 via the user input
9 interface 170 or other appropriate mechanism.

10
11 Depicted in Fig. 1, is a specific implementation of a WAN via the Internet.
12 Here, computer 130 employs modem 178 to establish communications with at
13 least one remote computer 182 via the Internet 180.

14
15 In a networked environment, program modules depicted relative to
16 computer 130, or portions thereof, may be stored in a remote memory storage
17 device. Thus, e.g., as depicted in Fig. 1, remote application programs 189 may
18 reside on a memory device of remote computer 182. It will be appreciated that the
19 network connections shown and described are exemplary and other means of
20 establishing a communications link between the computers may be used.

21 22 Converters and Lazy Loading

23 To enhance portability of programming languages and compiled codes,
24 methods and/or converters are presented herein to perform the following acts: (i)
25 compile a programming language code associated with a first framework (e.g., a

bytecode framework) to a compiled code associated with a second framework (e.g., an IL code framework); and/or (ii) convert a compiled code associated with a first framework (e.g., a bytecode framework) to a compiled code associated with a second framework (e.g., an IL code framework). The exemplary methods and/or converters operate in conjunction with loaders, such as, but not limited to, class loaders. An exemplary method, described herein, receives an initial code associated with a first framework, the initial code including a reference to a referenced class; converts the initial code to a converted code capable of execution on a second framework; executes the converted code on the second framework; detects a need for the referenced class during execution of the converted code on the second framework; and loads the referenced class into memory assessable by the second framework.

Fig. 2 shows a block diagram of an exemplary user system 200 and an exemplary Web site and/or server 340. The user system 200 includes various components of the exemplary computer 120 and peripheral system shown in Fig. 1. The user system 200 and the Web site/server 340 are connected electronically, for example, via a network such as a LAN, a WAN, etc. As shown in Fig. 2, the Web site/server 340 includes classes 344 while the user system 200 includes a Web browser 300, an applet hosting control 310, a converter 400 and a framework 500, such as the .NETTM framework. The converter 400 of the user system 200 can convert a bytecode to an IL code, such as, a JAVATM language framework bytecode to a .NETTM framework IL code. Alternatively, the converter 400 can compile a programming language code to a compiled code, such as, a JAVATM language framework programming language code to a .NETTM framework IL

1 code. The user system 200 has the ability to receive bytecode (or programming
2 language code), convert the bytecode (or programming language code) to IL code,
3 and execute the IL code on a framework different from the bytecode's (or
4 programming language code's) associated framework.

5
6 Referring to Fig. 2, in general, the Web site/server 340 comprises a Web
7 page server application for hosting a Web page (e.g., Web page 304) on a Web
8 browser (e.g., Web browser 300). Such an application is capable of transmitting
9 other applications in bytecode and/or IL code to a Web browser.

10
11 In the JAVATM language framework, an application in bytecode, or a
12 bytecode application, is typically referred to as an "applet". An applet generally
13 comprises a small, self-contained computer program that performs a task or tasks
14 as part of, or under the control of, a larger software application. For example,
15 most modern World Wide Web browsers are capable of making use of applets
16 written in a JAVATM programming language to perform tasks such as displaying
17 animations, operating spreadsheets and/or databases, etc. As described herein, the
18 term "applet" is not limited to computer programs written in a JAVATM
19 programming language and/or compiled to JAVATM language bytecode.

20
21 When a Web browser encounters an "applet" tag in a Web page, an applet
22 class loader is normally invoked. A JAVATM language framework associated
23 applet class loader is typically a JAVATM language class that contains code for
24 fetching an applet's executable code (e.g., bytecode) and classes referenced by the
25 executable code. In the JAVATM language framework, classes are defined in a

1 machine-independent, binary representation known as the class file format. A
2 class usually comprises a small unit of a JAVATM language framework software
3 component. In addition, an individual class representation is called a class file
4 even though it need not be stored in an actual file. For example, class files can be
5 stored as records or commands in a database. A class file can contain bytecode as
6 well as symbolic references to fields, methods, and names of other classes.

7
8 A typical JAVATM language framework bytecode includes classes that may
9 be required for execution of an applet. Execution of bytecode normally involves
10 identifying a reference to a class and checking to see if that class has already been
11 loaded into memory. The class referenced, is referred to herein as a referenced
12 class. If the referenced class has not been loaded, then the loader attempts to load
13 the referenced class, usually first from a local disk and thereafter from a Web
14 site/server. Thus, in the JAVATM language framework, classes are loaded on
15 demand or “just-in-time” during bytecode execution. This form of class loading is
16 referred to herein as “lazy loading”.

17
18 Referring again to Fig. 2, the applet hosting control 310 includes an applet
19 class loader 320 that can fetch an applet’s bytecode and identify and/or fetch
20 referenced classes. In the exemplary user system 200 shown in Fig. 2, the applet
21 class loader 320 can fetch referenced classes from a local source and/or a remote
22 source such as the Web site/server 340, for example, from the classes 344 on the
23 Web site/server 340. While the applet hosting control 310 and the applet class
24 loader 320 optionally comprise JAVATM language framework components, this is
25 not a requirement. In particular, a JAVATM language RE is not required because

1 the user system 200 includes a framework RE 520 such as the .NET™ framework
2 RE, which optionally has its own associated applet class loader. Alternatively, an
3 applet hosting control 310, having an applet class loader 320, runs on the user
4 system 200, for example, in conjunction with Web browser software and/or as a
5 separate application that runs on an operating system of the user system 200.

6
7 Overall, the exemplary user system 200, shown in Fig. 2, can receive code
8 associated with a first framework, convert it, and execute it on a second
9 framework 500. Furthermore, the applet hosting control 310 allows for lazy
10 loading of code associated with the first framework during execution of converted
11 code on the second framework. Fig. 3 shows a block diagram of an exemplary
12 method 600 for use with the exemplary user system 200 shown in Fig. 2 and
13 optionally other user systems. While the following description of Fig. 3 cross-
14 references Fig. 2, again, it is understood that the exemplary method 600 may be
15 practiced on other user systems.

16
17 Referring to Fig. 3, in a receiving block 610, a user system 200 receives an
18 applet associated with a first framework. Next, in a conversion block 614, a
19 converter 400 converts code associated with the applet to a converted code,
20 capable of execution on a second framework 500. In an execution block 618, a
21 RE 520 associated with the second framework 500 executes the converted code.
22 At some point in time during execution, in a detection block 622, the RE 520
23 detects a need for a referenced class, i.e., a class referenced by the applet and/or
24 converted code. Subsequently, in a loading block 626, a loader 320 fetches (i.e.,
25 locates and/or loads) the referenced class. Thereafter, in another conversion block

630, the converter 400 converts code associated with the referenced class to code capable of execution on the framework 500. Next, an execution block 634 resumes execution of the applet's converted code and/or referenced class's converted code on the RE 520 (generally referred to herein as applet execution).

The exemplary user system 200, shown in Fig. 2, optionally includes an application domain and a dynamic assembly. In such a system, the framework 500 operates in conjunction with the applet hosting control 310 and the converter 400 to allow for code associated with a first framework to execute on a second framework, e.g., framework 500. In particular, the converter 400 (and/or optionally the framework 500) provides for the creation of an application domain that maintains a dynamic assembly of classes or types for code referenced and/or converted by the converter 400.

An application domain is an isolated environment where applications execute. In an exemplary framework (e.g., framework 500), an application domain class provides methods for performing the following tasks: enumerating assemblies in a domain; defining dynamic assemblies in a domain; specifying assembly loading and domain termination events; loading assemblies and types into the domain; and terminating the domain. In general, an RE (e.g., RE 520) manages all memory in an application domain.

Fig. 4 illustrates a block diagram of another exemplary method 700. While the following description of Fig. 4 cross-references Fig. 2, again, it is understood that the exemplary method 700 may be practiced on other user systems. Referring

1 to Fig. 4, in a loading block 710, the Web browser 300 loads a Web page 304 from
2 a Web site/server 340. Upon loading the Web page 304, the Web browser 300
3 checks for applets in a check block 714 wherein the Web browser 300 encounters
4 an applet with a class file Class X1.

5
6 Class file Class X1 contains the following exemplary code:

```
7  
8 public ClassX1 extends Applet  
9 {  
10     public void MyMethod1( )  
11     {  
12         ...  
13         new ClassX2( ).bar( );  
14         ...  
15     }  
16  
17     public void MyMethod2( )  
18     {  
19         ...  
20         ClassX3 x = new ClassX3( );  
21         ...  
22     }  
23 }  
24  
25
```

18 In response to encountering the applet, the Web browser 300, through an
19 invocation block 718, invokes an applet hosting control 310. In turn, the applet
20 hosting control 310, through another invocation block 722, invokes an applet class
21 loader 320. Subsequently, in a load block 726, the applet class loader 320 loads
22 the class file for ClassX1. In general, the applet class loader 320 loads class files
23 from a remote site or server, such as Web site/server 340, via HTTP. In Fig. 2,
24 Web site/server 340 includes class files 344, e.g., designated X1, X2, X3,
25

1 The class files include code associated with a particular framework, for example,
2 bytecode associated with the JAVATM language framework.

3
4 Once the applet class loader 320 has loaded ClassX1 bytecode, e.g.,
5 bytecode 324, a converter 400 in a conversion block 730 converts the bytecode
6 324 to IL code having an associated type. Thereafter, the converter 400 (and/or
7 optionally the framework 500), through a definition block 734, defines a dynamic
8 assembly in an application domain. In the .NETTM framework, an assembly is a
9 collection of types and resources that are built to work together and form a logical
10 unit of functionality. The .NETTM framework supports static and dynamic
11 assemblies. A .NETTM framework typically creates a dynamic assembly using
12 reflection emit APIs; use of APIs for dynamic assemblies and/or conversion is
13 described in more detail in a separate section below. To the .NETTM framework's
14 RE, a type does not exist outside the context of an assembly.

15
16 Once a dynamic assembly has been defined, the converter 400 (and/or
17 optionally the framework 500), in an instantiation block 738, instantiates the type
18 for ClassX1 in the dynamic assembly. In some systems, ClassX1 may now be
19 referred to as being "baked" because it is ready for execution by the framework's
20 RE. However, referring to the aforementioned code for class file ClassX1, this
21 code includes references to ClassX2 and ClassX3. Since these classes have not
22 been instantiated as types in the dynamic assembly, they are not ready for
23 execution by the framework's RE. In some systems, ClassX2 and ClassX3 may be
24 referred to as being "unbaked".
25

1 Referring again to the exemplary user system 200 shown in Fig. 2 and the
2 exemplary method 700 shown in Fig. 4, the converter 400 (and/or optionally the
3 framework 500), through a reference block 742, emits type references for classes
4 ClassX2 and ClassX3. Thus, in addition to the type for ClassX1, the dynamic
5 assembly now contains type references for class files that may be needed for
6 execution of the ClassX1 applet.

7
8 In an execution block 746, the RE 520 executes the ClassX1 applet. The
9 execution continues until the RE 520 encounters an unresolved type. For example,
10 in an execution block 750, the RE 520 encounters an unresolved reference for type
11 ClassX2 in the dynamic assembly. In response to this encounter, the application
12 domain, in an invocation block 754, invokes an event resolver, which invokes the
13 applet class loader 320 and optionally a handler.

14
15 In a load block 758, the applet class loader 320 searches for the class file
16 for ClassX2 and fetches the file, for example, from the Web site/server 340. The
17 conversion process, as already described, uses the converter 400 to convert,
18 through a conversion block 762, the ClassX2 bytecode to IL code. Again, type
19 references are created if ClassX2 references additional classes. After the
20 conversion, in a return and continuation block 766, the handler returns and/or
21 gives notification that the conversion for ClassX2 has taken place and the
22 converter 400 (and/or optionally the framework 500), instantiates type ClassX2 in
23 the dynamic assembly, and the framework 500 continues execution of the ClassX1
24 applet.
25

1 As described, the exemplary method 700 shown in Fig. 4 relies on lazy
2 loading. In particular, ClassX2 was not loaded or converted until an impending
3 need for ClassX2 was shown to exist, or detected, during execution of the ClassX1
4 applet. Lazy loading can reduce memory usage because certain classes referenced
5 by an applet may never be called during execution of the applet. To load all
6 classes, regardless of whether they are called or not, would introduce an up-front
7 demand requiring both time for loading and conversion and memory for storing
8 loaded and converted code.

9
10 While Fig. 4 shows an exemplary method 700 using lazy loading, the
11 exemplary user system 200 shown in Fig. 2 has the ability to perform up-front
12 loading with conversion on a lazy basis, i.e., "eager" loading and lazy conversion.
13 In addition, this exemplary system 200 may also perform eager loading and
14 conversion with lazy instantiation of types and/or type references into the dynamic
15 assembly.

16
17 The exemplary user system 200 and the exemplary method 700 of Figs. 2
18 and 4, respectively, allow code associated with a first framework to execute on a
19 second framework. When the first framework relies on lazy loading, the second
20 framework should also rely on lazy loading to mimic operation of the first
21 framework and to retain the programmer's original intent.

22
23 In another exemplary user system, a Web browser is optional. In such a
24 system, a user does not interact with an applet via a browser. For example, the
25 user system may comprise a server or other computer-enabled device wherein the

1 applet executes automatically without any direct human user input. For this
2 exemplary system, as for the others mentioned herein, a user refers to a human
3 user and/or a device user. The same definition of user also applies to exemplary
4 methods described herein.

6 Use of APIs for Conversion and/or Lazy Loading

7 As mentioned above, the exemplary system 200 and/or exemplary method
8 700 of Figs. 2 and 4, respectively, optionally use APIs for dynamic assemblies
9 and/or conversion. In particular, as described above, during execution of an
10 applet, an RE may encounter a type or a class that is not instantiated or referenced
11 in the dynamic assembly. An event resolver and/or a handler act as hooks to
12 fetch, convert, reference and/or instantiate the missing type or class.

13
14 In the case that a RE encounters a class that, for the time being, is only
15 referenced by an applet, then the framework invokes an API to generate an
16 incomplete type definition for the referenced class. The incomplete type
17 definition, or dummy class, only refers to a method or a property within the
18 referenced class. An API is used to emit a reference type corresponding to the
19 incomplete type. Again, in some systems, such a class or type is referred to as
20 being “unbaked”.

21
22 In the .NETTM framework, the hook optionally includes an event handler
23 for an OnResolveType event. In an exemplary method, during loading of a first
24 applet class, an event handler is registered using an AddOnTypeResolve method
25 on an assembly. If and when the code containing the reference is actually

1 executed, the OnResolveType event is raised and the AddOnTypeResolve handler
 2 code recognizes the event as a fault corresponding to a missing class or type. The
 3 handler then invokes an applet class loader, which attempts to fetch the missing
 4 class, and a converter, which converts the bytecode, metadata, class
 5 implementation, methods, inheritance properties, fields etc. into IL code and/or IL
 6 metadata. This conversion is optionally performed using reflection emit APIs, for
 7 example, the APIs in the .NETTM framework's System.Reflection.Emit namespace.
 8 For a given bytecode class, such APIs can create appropriate constructs of each
 9 method, field, interface in IL code, which can be emitted into the dynamic
 10 assembly.

11
 12 A .NETTM framework optionally includes the following APIs:
 13 ConstructorBuilder class, which defines and represents a constructor of a dynamic
 14 class and, in conjunction with TypeBuilder class, it can create classes at run time;
 15 FieldBuilder class, which defines and represents a field; MethodBuilder class,
 16 which defines and represents a method (or constructor) on a dynamic class and, in
 17 conjunction with TypeBuilder class, it can create classes at runtime; PackingSize
 18 Enumeration, which specifies the packing size of a type; TypeAttributes
 19 Enumeration, which specifies type attributes; FieldAttributes Enumeration, which
 20 specifies flags that describe the attributes of a field; MethodAttributes
 21 Enumeration, which specifies flags for method attributes; and TypeBuilder class,
 22 which defines and creates new instances of classes during runtime. The
 23 TypeBuilder class is the root class used to control the creation of dynamic classes
 24 and it provides a set of routines that are used to define classes, add methods and
 25 fields, and create the class inside a domain.

1
2 According to the exemplary method of Fig. 4, when a reference type (e.g.,
3 dummy or unbaked type) is created, various type characteristics are not necessarily
4 specified (e.g., Private, Public, LayoutSequential, Sealed, etc.). Thus, an
5 exemplary system and/or exemplary method includes a means to set and/or change
6 such characteristic once a loader fetches a class file corresponding to the reference
7 type and/or once a converter converts a class file bytecode, corresponding to the
8 reference type, to IL code. A suitable means optionally includes the following
9 methods in TypeBuilder: a method to change the TypeAttributes; and a method to
10 change the PackingSize.

11
12 In addition, for dummy methods inside a reference type (e.g., dummy or
13 unbaked), various method characteristics are not necessarily specified (e.g.,
14 Private, Public, Static, Final, PInvoke, etc.). Thus, an exemplary system and/or an
15 exemplary method include a means to set and/or change such characteristic. A
16 suitable means optionally includes the following methods in MethodBuilder: a
17 method to change the MethodAttributes; and a method to change an ordinary
18 method to a Platform Invoke (PInvoke) method. Further support includes a
19 method to set and/or change TypeAttributes of a ConstructorBuilder and/or
20 FieldAttributes of a FieldBuilder.

21
22 While various exemplary converters and methods described herein apply to
23 converting code from a JAVATM language framework to a .NETTM framework,
24 conversions from a .NETTM framework to a JAVATM language framework are also
25

1 within the scope of exemplary systems and exemplary methods presented herein
2 as well as conversions to, from and/or between other frameworks known in the art.
3

4 Thus, although some exemplary methods and exemplary systems have been
5 illustrated in the accompanying Drawings and described in the foregoing Detailed
6 Description, it will be understood that the methods and systems are not limited to
7 the exemplary embodiments disclosed, but are capable of numerous
8 rearrangements, modifications and substitutions without departing from the spirit
9 set forth and defined by the following claims.
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25